CP3-11-25, IPHC-PHENO-11-04, IPPP/11/39, DCPT/11/78, MPP-2011-68

UFO - The Universal FeynRules Output

Céline Degrande [a], Claude Duhr [b], Benjamin Fuks [c],
David Grellscheid [b], Olivier Mattelaer [a], Thomas Reiter [d]

[a] *Université catholique de Louvain,*
Center for particle physics and phenomenology (CP3),
Chemin du cyclotron, 2, B-1348 Louvain-La-Neuve, Belgium
Email: celine.degrande@uclouvain.be, olivier.mattelaer@uclouvain.be

[b] *Institute for Particle Physics Phenomenology, University of Durham,*
Durham, DH1 3LE, United Kingdom,
Email: claude.duhr@durham.ac.uk, david.grellscheid@durham.ac.uk

[c] *Institut Pluridisciplinaire Hubert Curien/Département Recherches Subatomiques,*
Université de Strasbourg/CNRS-IN2P3, 23 Rue du Loess, F-67037 Strasbourg,
France
E-mail address: benjamin.fuks@iphc.cnrs.fr

[d] *Max-Planck-Institut für Physik,*
Föhringer Ring 6, 80805 München, Germany
Email: reiterth@mpp.mpg.de

Abstract

We present a new model format for automatized matrix-element generators, the so-called *Universal FeynRules Output (UFO)*. The format is universal in the sense that it features compatibility with more than one single generator and is designed to be flexible, modular and agnostic of any assumption such as the number of particles or the color and Lorentz structures appearing in the interaction vertices. Unlike other model formats where text files need to be parsed, the information on the model is encoded into a Python module that can easily be linked to other computer codes. We then describe an interface for the Mathematica package FeynRules that allows for an automatic output of models in the UFO format.

Key words: Model building, model implementation, Feynman rules, Feynman diagram calculators, Monte Carlo programs.

1 Introduction

Monte Carlo simulations of the physics to be observed at the Large Hadron Collider (LHC) at CERN play a central role in the exploration of the electroweak scale, both from the experimental point of view of establishing an excesses over the expected Standard Model (SM) backgrounds as well as from the phenomenological point of view by providing possible explanations for the observations. For this reason, activities in the field of Monte Carlo simulations have been rather intense over the last fifteen years, resulting in many advances in the field. Automated tree-level matrix-element generators, such as ALPGEN [1], COMIX [2], COMPHEP/CALCHEP [3,4,5], HELAC [6], HERWIG [7,8], MADGRAPH/MADEVENT [9,10,11,12,13], SHERPA [14,15], or WHIZARD [16,17], describing the hard scattering processes where the Beyond the Standard Model (BSM) physics is expected to show up have been developed. As a consequence, the problem of the automatic generation of tree-level matrix elements for a large class of Lagrangian-based BSM theories is solved, at least in principle.

Due to the numerous existing BSM theories based on ideas in constant evolution, the implementation of these models into Monte Carlo event generators remains a tedious and error-prone task. Feynman rules associated with a given BSM model must be derived and then implemented one at the time into the various codes, which often follow their own specific conventions and formats. A first step in the direction of automating this procedure by starting directly from the Lagrangian of the model has been made in the context of the LANHEP package [18] linked to the COMPHEP and CALCHEP programs. Recently, a new efficient framework going beyond this scheme has been developed. It is based on the FEYNRULES package [19,20,21,22] and proposes a general and flexible environment allowing to develop a model, investigate its phenomenology and eventually confront it to data. Its virtue has been illustrated in the context of the COMPHEP/CALCHEP, FEYNARTS/FORMCALC [23], MADGRAPH/MADEVENT, SHERPA and WHIZARD programs, by implementing several new physics theories in FEYNRULES and then passing them to the different tools for a systematic validation procedure. The approach is based on a modular structure where each node consists in an interface to a dedicated matrix-element generator. Since the latter have in general hard-coded information regarding the supported Lorentz and/or color structures, the interfaces check whether a given vertex is compliant with a given matrix-element generator, in which case the vertex is written to file in a format suitable for the generator. The final output consists then in a set of text files that can be used in a similar way to any other built-in model.

The procedure spelled out above, where communication between FEYNRULES and the matrix-element generators proceeds exclusively via a set of well-

defined text files that must be parsed and interpreted, has some serious limitations. In particular, extending the format to include more general structures, like higher-dimensional operators and/or non-standard color structures, is difficult to incorporate into a static text-based format. In this paper we present a new format, dubbed the Universal FeynRules Output or the UFO, for model files that goes beyond existing formats in various ways. The format is completely generic and, unlike existing formats, it does not make any *a priori* assumptions on the structures that can appear in a model. The aim is to provide a flexible format, where *all* the information about a model is represented in an abstract form that can easily be accessed by other tools. The information on the particles, parameters and vertices of the model are stored in a set of Python objects, each of them being associated with a list of attributes related to their properties. This way of representing the model information has some benefits over the more traditional plain text table-based format, because it allows, *e.g.*, to add a missing piece of information directly as a new attribute to an existing object. As an example, extending a table-based format to accommodate higher-point vertices requires to change the format of the table and to adapt the readers for the table accordingly. In an object-oriented format like the UFO, the same extension is trivial, as the number of particles entering a vertex is just an attribute of the vertex, so no extension of rewriting of the readers is necessary. Presently, the UFO format is already used by the MadGraph version 5 [13] and the GoSam generators [24,25,26], and will be used in a near future by Herwig++.

The paper is organized as follows. In Section 2, we describe the features of the UFO format as a stand-alone Python module, while Section 3 addresses the automation of writing an UFO model through FeynRules. Section 4 is dedicated to the UFO features beyond tree level and in Section 5, we provide an example of how to implement a model containing non-trivial Lorentz structures with the help of FeynRules into MadGraph 5. Our conclusions are drawn in Section 6.

2 The UFO format

Any quantum field theory can be defined by a threefold information,

- a set of particles, defined together with their quantum numbers (spin, electric charge, *etc.*),
- a set of parameters (masses, coupling constants, *etc, ...*),
- a Lagrangian describing the interactions among the different particle species.

However, matrix-element generators do not work, in general, directly with the Lagrangian, but rather with an explicit set of vertices. In the rest of this

section, we assume that we have extracted all the vertices from the Lagrangian of a given model and only restrict ourselves to describing a new generic format to implement the information on the particles and parameters of the model along with the vertices describing the interactions among the particles into matrix-element generators. The issue of the extraction of the vertices from the Lagrangian and their translation into this new format in an automated fashion via the FeynRules package will be discussed in Section 3.

The Universal FeynRules Output (UFO) allows to translate all the information about a given particle physics model into a Python module that can easily be linked to existing matrix-element generators. While in general each generator is following its own format and conventions, the UFO format goes beyond this approach in the sense that it is, by definition, not tied to any specific matrix-element generator. More specifically, it saves the model information in an abstract (generator-independent) way in terms of Python objects. An UFO model is hence a standalone Python module, containing ready-to-go definitions for all the classes representing particles, parameters, *etc.*, and which can be directly linked to an existing matrix-element generator without any modification or further interfacing.

In this section we give a detailed account on the UFO format, putting special emphasis on the definition of the different classes useful for designing model files. In general, an UFO model consists of a directory containing a set of text files that can be split into two distinct classes,

- Model-independent files:
  - `__init__.py`,
  - `object_library.py`,
  - `function_library.py`,
  - `write_param_card.py`,
- Model-dependent files:
  - `particles.py`,
  - `parameters.py`,
  - `vertices.py`,
  - `couplings.py`,
  - `lorentz.py`,
  - `coupling_orders.py`.

Since the UFO format is based on the Python language, all files have a `.py` extension. The model-independent files are identical for every model and contain, among others, the definitions of the classes which the model-dependent objects (particles, parameters, *etc.*) are instances of. All those files are provided as self-contained Python modules.

A file named `__init__.py` inside a directory is standard in the PYTHON language and corresponds to a tag for importing complete PYTHON modules by issuing the PYTHON command

```
import Directory_Name
```

where `Directory_Name` refers to the name of the directory containing the `__init__.py` file. However, in addition to the possiblity of importing a complete UFO model, this file also contains, in the UFO case, links to different lists of quantities associated with the various objects defined in a model,

- `all_particles`
- `all_vertices`
- `all_couplings`
- `all_lorentz`
- `all_parameters`
- `all_coupling_orders`
- `all_functions`

These lists allow, *e.g.*, to access the full particle content of a model in an easy way downstream in the code. Moreover, every time that an instance of a class is created in the model, it will be automatically added to the corresponding list.

An UFO model can be fully implemented with the help of a small number of basic classes, denoted `Particle`, `Parameter`, `Vertex`, `Coupling`, `Lorentz` and `CouplingOrder`. All of these classes are derived from the mother class `UFOBaseClass`, defining a set of common methods and attributes accessible in the same way by each class. The mother class, together with all its children, is defined in the file `object_library.py`. In particular, each class has methods to display all the attributes associated to a given instance of the class, as well as to return or set the values of these attributes. As an example, if `P` is an instance of the class `Particle` and if `charge` is an attribute of this class (see Section 2.2), the charge of the corresponding particle can be accessed in the standard way by issuing the command `P.charge`. The complete list of attributes of the `UFOBaseClass` class is summarized in Table 1.

The file `function_library.py` is related to the implementation of user-defined functions into an UFO module via the special class `Function`, which translates functions that can be defined within a single PYTHON line (*i.e.*, the so-called PYTHON 'lambda' functions) to other programming languages (such as FORTRAN or C++). Let us note that this specific type of functions is currently the only type of user-defined functions supported by the UFO format. A mem-

Table 1: Attributes and methods available to all UFO classes

`get_all`	Returns a list of all the attributes of an object.
`nice_string`	Returns a string with a representation of an object containing the values associated with each of its attributes.
`get`	This method provides access to the value of an attribute of an object. As an example, if `P` denotes an instance of a class with attribute `charge`, then `P.get('charge')` and `P.charge` are equivalent means of accessing the value of the attribute `charge`.
`set`	This method allows one to modify the value of an attribute of an object. As an example, if `P` denotes an instance of class with attribute `charge`, then `P.set('charge', 1)`, or equivalently `P.charge = 1`, will set the attribute denoted by `charge` to unity.

ber of the class `Function` contains three mandatory attributes, called `name`, `arguments` and `expression`. While `name` is a string representing the name of the function, the attributes `arguments` and `expression` correspond to a sequence of strings for the names of the variables the function depends upon and a string representing the valid PYTHON expression defining the function itself. Several functions are by default included into the UFO function library,

- `complexconjugate`: complex conjugation,
- `csc`: the trigonometric function cosecant,
- `acsc`: the cyclometric function arccosecant,
- `im`: the imaginary part of a complex number,
- `re`: the real part of a complex number,
- `sec`: the trigonometric function secant,
- `asec`: the cyclometric function arcsecant.

These functions consist in a set of common mathematical functions for which the standard PYTHON module `cmath` is insufficient or unpractical. As an example, the cosecant function `csc` (not included in the `cmath` library) is implemented within the UFO module as an instance of the aforementioned class `Function` via

```
csc = Function(name = 'csc',
               arguments = ('z',),
               expression = '1./cmath.sin(z)')
```

In the UFO format, all particles are instances of the class `Particle` defined in the file `particles.py`. Even if the Lagrangian of a model is in general more easily written in terms gauge eigenstates, matrix-element generators usually work at the level of mass eigenstates. Hence only mass eigenstates should be defined in the `particles.py` file.

The definition of a particle might read, for, *e.g.*, a top quark, as

```
t = Particle( pdg_code = 6,
              name = 't',
              antiname = 't~',
              spin = 2,
              color = 3,
              mass = Param.MT,
              width = Param.WT,
              texname = 't',
              antitexname = '\\bar{t}',
              charge = 2/3,
              line = 'straight',
              LeptonNumber = 0
              )
```

The class `Particle` has various attributes that are summarized in Table 2. In the following we content ourselves to highlight only the most important points. First, note that, apart from a set of mandatory arguments (all attributes but the last two in the example above), the `Particle` class can be given an arbitrary number of optional attributes (the `line` and `LeptonNumber` attributes in the example). There are three predefined optional attributes, which are summarized in Table 2. Every additional optional attribute must be an integer representing additional model-dependent additive quantum numbers (as the attribute `LeptonNumber` in the example). The only exceptions regarding the treatment of the quantum numbers concern the electric charge and color representation, which are always mandatory and stored in the attributes `charge` and `color`.

A particle object is identified through its name, a string stored in the `name` attribute. In a similar fashion, the attribute `antiname` is a string representing the name of the corresponding antiparticle. Note that self-conjugate particles, *i.e.*, particles that are their own antiparticles, are identified by having identical `name` and `antiname` attributes (*i.e.*, even for self-conjugate particles, the `antiname` attribute must be defined). The transformation properties of the particle under the Lorentz group and the QCD and electromagnetic gauge

groups are specified through the `spin`, `color` and `charge` attributes. Each of these attributes takes an integer value:

- `spin`: the possible values are $2s+1$, where s is the spin of the particle. For the moment only $s \leq 2$ is supported. By convention, the value -1 is used for ghost fields.
- `color`: the possible values are 1, ± 3, ± 6 and 8, corresponding to singlets, (anti)triplets, (anti)sextets and octets.
- `charge`: any rational number, representing the electric charge of the particle.

Inside matrix-element generators, particles are often identified through an integer number referring to the Particle Data Group (PDG) numbering scheme [27]. This code is stored in the `pdg_code` attribute, which can be set to any integer value, even though it is highly recommended to follow the existing conventions whenever possible. Finally, masses and widths are encoded in the `mass` and `width` attributes. They refer to the corresponding instances of the `Parameter` class defined in the file `parameters.py` (see Section 2.3). Therefore, at the beginning of the `particles.py` file, the `Parameter` objects are imported via the PYTHON instruction

```
import parameters as Param
```

In the previous example we have only instantiated the object representing the top quark. However, since the top quark is not a self-conjugate particle, we still need to implement an object representing the top antiquark. We could proceed in a similar way as in the example above, but the `Particle` class has a built-in method, denoted `anti()`, instantiating the antiparticle object directly from the corresponding particle object. In the example of the top quark, the instruction

```
t__tilde__ = t.anti()
```

instantiates a `Particle` object called `t__tilde__` which is identical to the object `t` previously defined, but with the attributes `name` (`texname`) and `antiname` (`antitexname`) interchanged. In addition, all the quantum numbers, including the electric charge (`charge`) and the color representation (`color`), are set to opposite values.

2.3 Implementation of the parameters of a model

Parameters of a model, like masses, coupling constants, *etc.*, are defined in an UFO model as instances of the `Parameter` class (itself defined in the file `object_library.py`) in the file `parameters.py`. All the parameters used in a model implementation are either *external* (or equivalently *independent*) pa-

Table 2: Attributes of the particle class

`pdg_code`	An integer corresponding the identification number related to the PDG numbering scheme [27].
`name`	A string specifying the name of the particle.
`antiname`	A string specifying the name of the antiparticle. If the particle is self-conjugate, `antiname` must be identical to `name`.
`spin`	An integer corresponding to the spin of the particle in the form $2s + 1$. By convention, the spin of a ghost field (anti-commuting scalar field) is -1.
`color`	An integer corresponding to the dimension of the color representation of the particle $(1, \pm 3, \pm 6, 8)$.
`mass`	A `Parameter` object corresponding to the mass of the particle. If the particle is massless, the value must be set to `Param.ZERO`.
`width`	A `Parameter` object corresponding to the width of the particle. If the width is zero, the value must be set to `Param.ZERO`.
`texname`	A TEX string representing the particle name.
`antitexname`	A TEX string representing the antiparticle name.
`charge`	A rational number equal to the electric charge of the particle.

Optional attributes

`goldstone`	A boolean, tagging a scalar field as a Goldstone boson (`true`) or not (`false`). The default value is `false`.
`propagating`	A boolean, tagging the corresponding particle as auxiliary and non-propagating (`false`) field or as a physical field (`true`). The default value is `true`.
`line`	A string representing how the propagator of the particle should be drawn in a Feynman diagram. The possible values are `'dashed'`, `'dotted'`, `'straight'`, `'wavy'`, `'curly'`, `'scurly'`,`'swavy'` and `'double'`. The default value is chosen according to the spin and color representation of the particle.

rameters or *internal* (or equivalently *dependent*) parameters. The user must provide as an input the numerical value of the external parameters (*e.g.*, $\alpha_s = 0.118$), while the internal parameters are related to other (external and/or internal) parameters via algebraic relations (*e.g.*, $g_s = \sqrt{4\pi\alpha_s}$). Since internal and external parameters belong to the same generic class `Parameter`, their declaration is very similar. We will give an example for each case separately in order to emphasize the main differences and features. The list of all the possible attributes for the `Parameter` class is summarized in Table 3.

We start with external parameters. In the UFO format, the external parameters are all taken to be real and the `type` attribute of the `Parameter` class must be set to the value `'real'`. Therefore, complex numbers will have to be split into their real and imaginary parts. As an example, the declaration of the external parameter α_s reads

```
aS = Parameter(name = 'aS',
               nature = 'external',
               type = 'real',
               value = 0.118,
               texname = '\\alpha_s',
               lhablock = 'SMINPUTS',
               lhacode = [ 3 ]
               )
```

The attributes of the `Parameter` class are all mandatory and contain the name of the parameter (`name`), its nature (`nature`) which is external and the value of the parameter (`value`). Since any external parameter is a *real* number, the value must be a real floating point number. The last two arguments, `lhablock` and `lhacode`, refer to the Les Houches-like format for the input parameters which is followed by the UFO. This is a generalization to any model of the original Supersymmetry Les Houches Accord [28,29]. All the model parameters are grouped into blocks, each line of a block containing a counter (a sequence of integers) associated with a given parameter name and its corresponding numerical value. The attribute `lhablock` of the `Parameter` object directly refers to the name of the block in which the considered parameter is stored, whilst the attribute `lhacode` is a list of integers referring to the counter.

An additional function related to the Les Houches format is included in the file `write_param_card.py`. The class `ParamCardWriter` can be called from within another PYTHON module by issuing the instruction

```
ParamCardWriter('./param_card.dat', qnumbers=True)
```

and outputs a parameter file named `param_card.dat` which contains all the external parameters defined in the model, grouped into blocks and counters according to their `lhablock` and `lhacode` attributes. The first argument in

the function above refers to the location of the output file, whereas the second argument specifies whether or not the QNUMBERS blocks [30] should be included in the output. In addition, if the second argument is set to True, the full set of masses and widths, even if they are dependent parameters, are written to file. In the example of aS presented above, the corresponding entry in the output file would read

```
Block SMINPUTS
    3 1.18000e-01 # aS
```

Let us also note that the file write_param_card.py can be directly used from the command line by issuing the instruction

```
$> python ./write_param_card.py
```

As a result, an output file named param_card.dat is created and contains the numerical values of all the external parameters. A snapshot of this parameter file for a more complete model reads

```
###################################
## INFORMATION FOR SMINPUTS
###################################
Block SMINPUTS
    1 1.325070e+02 # aEWM1
    2 1.166390e-05 # Gf
    3 1.180000e-01 # aS

###################################
## INFORMATION FOR YUKAWA
###################################
Block YUKAWA
    5 4.200000e+00 # ymb
    6 1.645000e+02 # ymt
   15 1.777000e+00 # ymtau
```

The definition of internal parameters follows the same lines as for the external parameters, with the only differences that the lhablock and lhacode attributes are not available and that the value argument now contains an algebraic expression relating the parameter to other external or internal parameters. As a simple example, consider the external parameter aS (α_s) and the internal parameter G ($g_s = \sqrt{4\pi\alpha_s}$). The implementation of G reads,

```
G = Parameter(name = 'G',
              nature = 'internal',
              type = 'real',
              value = 'cmath.sqrt(4 * cmath.pi * aS)',
```

Table 3: Parameter class attributes

`nature`	A string, either `'external'` or `'internal'`, specifying whether a given parameter is considered as a dependent or independent parameter.
`name`	A string, specifying the name of the parameter.
`type`	A string, either `'real'` or `'complex'`, specifying whether a given parameter is a real or a complex number. We remind that following the UFO synthax, external parameters must be real numbers.
`value`	For external parameters, this attribute is a single real floating-point number. For internal parameters, it consists of a string representing the analytic expression relating the considered parameter to other external and/or internal parameters, following a valid PYTHON syntax.
`texname`	A TEX string representing the parameter name in TEX format.
Attributes specific to external parameters	
`lhablock`	A string containing the name of the block which the parameter is assigned to, following a Les Houches-like format.
`lhacode`	A list of integers giving the position of the considered parameter inside a given `lhablock`, *i.e.*, the counter associated with the parameter, following a Les Houches-like format.

```
texname = 'G'
)
```

Unlike the case of external parameters, the `value` attribute is a string representing a valid algebraic PYTHON expression. Moreover, it is mandatory that every internal parameter depends only on other parameters which have already been declared. Returning to our example, the external parameter `aS` must hence be defined *before* the internal parameter `G` inside the file `parameters.py`. Note that masses and widths are considered to be parameters of the model (either internal or external), and must thus be declared as such in `parameters.py`.

Let us conclude this section by mentioning that most matrix-element generators have information on the Standard Model input parameters hard-coded. This allows, among others, for a correct handling of the running of the strong

coupling constant. Therefore, the Standard Model parameters in an UFO model must be correctly identified, following the same notations and conventions as for the implementation of a model in FeynRules [20].

2.4 Implementation of the interactions of the model

The vertices corresponding to the interactions included in a model are defined in the file `vertices.py` using the `Vertex` class. Let us consider a set of n particles $\{\phi_i^{\ell_i a_i}\}$, with spin indices [1] $\{\ell_i\}$ and color indices $\{a_i\}$. A generic n-point vertex coupling these fields can be written as a tensor in the color $\otimes$ spin space [2],

$$\mathcal{V}^{a_1\ldots a_n,\ell_1\ldots\ell_n}(p_1,\ldots,p_n) = \sum_{i,j} C_i^{a_1\ldots a_n}\, G_{ij}\, L_j^{\ell_1\ldots\ell_n}(p_1,\ldots,p_n)\ , \tag{2.1}$$

where the variables p_i denote the particle momenta, G_{ij} the couplings, and the quantities $C_i^{a_1\ldots a_n}$ and $L_j^{\ell_1\ldots\ell_n}(p_1,\ldots,p_n)$ denote tensors in color and spin space, respectively. Since several vertices may share the same spin and/or color tensors, the latter act as a 'basis' for the vertices of the model, the couplings being the 'coordinates' in that basis. As an example, the well-known four-gluon vertex,

$$\begin{aligned} & ig_s^2\, f^{a_1a_2b} f^{ba_3a_4}\ (\eta^{\mu_1\mu_4}\eta^{\mu_2\mu_3} - \eta^{\mu_1\mu_3}\eta^{\mu_2\mu_4}) \\ & \quad + ig_s^2\, f^{a_1a_3b} f^{ba_2a_4}\ (\eta^{\mu_1\mu_4}\eta^{\mu_2\mu_3} - \eta^{\mu_1\mu_2}\eta^{\mu_3\mu_4}) \\ & \quad + ig_s^2\, f^{a_1a_4b} f^{ba_2a_3}\ (\eta^{\mu_1\mu_3}\eta^{\mu_2\mu_4} - \eta^{\mu_1\mu_2}\eta^{\mu_3\mu_4})\ , \end{aligned} \tag{2.2}$$

is written in Eq. (2.1) as

$$\begin{aligned} & \left(f^{a_1a_2b} f^{ba_3a_4},\ f^{a_1a_3b} f^{ba_2a_4}, f^{a_1a_4b} f^{ba_2a_3}\right) \\ & \qquad \times \begin{pmatrix} ig_s^2 & 0 & 0 \\ 0 & ig_s^2 & 0 \\ 0 & 0 & ig_s^2 \end{pmatrix} \begin{pmatrix} \eta^{\mu_1\mu_4}\eta^{\mu_2\mu_3} - \eta^{\mu_1\mu_3}\eta^{\mu_2\mu_4} \\ \eta^{\mu_1\mu_4}\eta^{\mu_2\mu_3} - \eta^{\mu_1\mu_2}\eta^{\mu_3\mu_4} \\ \eta^{\mu_1\mu_3}\eta^{\mu_2\mu_4} - \eta^{\mu_1\mu_2}\eta^{\mu_3\mu_4} \end{pmatrix}\ . \end{aligned} \tag{2.3}$$

The UFO format for vertices mimics exactly this structure. As an example, the implementation of the vertex in Eq. (2.3) into an UFO model reads

```
V_1 = Vertex(name = 'V_1',
             particles = [ P.G, P.G, P.G, P.G ],
             color = [ 'f(1,2,-1)*f(-1,3,4)',
```

[1] The terminology *spin indices* refers to both Lorentz and Dirac indices.
[2] The case of non-strongly interacting particles corresponds to a tensor in color space equal to unity.

Table 4: Vertex class attributes

`name`	A string specifying the name tag of the vertex.
`particles`	A list of `Particle` objects containing the set of particles entering into the vertex. By convention, all particles are considered outgoing.
`color`	A list of strings representing the color tensors associated with the vertex, written as a polynomial combination of the elementary tensors given in Table 5.
`lorentz`	A list of `Lorentz` objects representing the spin tensors associated with the vertex.
`couplings`	A list of `Coupling` objects associated with the decomposition of the vertex in the color $\otimes$ spin space.

```
          'f(1,3,-1)*f(-1,2,4)',
          'f(1,4,-1)*f(-1,2,3)'  ],
lorentz = [ L.VVVV1, L.VVVV2, L.VVVV3 ],
couplings = {(0,0):C.GC_1,
             (1,1):C.GC_1,
             (2,2):C.GC_1}
)
```

The `Vertex` class is probably one of the most important features of the UFO format, since the vertices associated with a Lagrangian are at the heart of every implementation of a BSM model into a matrix-element generator. It requires five arguments, which are summarized in Table 4. First, each vertex is identified by an identification tag, its `name`. Next, the attribute `particles` contains the list of all `Particle` objects entering the considered vertex (by convention, all particles are considered outgoing). Since these objects are defined in the file `particles.py`, it is necessary to issue at the beginning of the file `vertices.py` the command

```
import particles as P
```

and a particle object `G` is now referred to as `P.G`. The attributes `color` and `lorentz` contain two lists with the color and Lorentz tensors associated with the vertex, *i.e.*, the quantities $C_i^{a_1\ldots a_n}$ and $L_j^{\ell_1\ldots\ell_n}(p_1,\ldots,p_n)$ appearing in Eq.

Table 5: Elementary color tensors

Trivial tensor (for non-colored particles)	`1`
Kronecker delta $\delta^{\bar{j}_2}{}_{i_1}$	`Identity(1,2)`
Fundamental representation matrices $(T^{a_1})^{\bar{j}_3}{}_{i_2}$	`T(1,2,3)`
Structure constants $f^{a_1a_2a_3}$	`f(1,2,3)`
Symmetric tensor $d^{a_1a_2a_3}$	`d(1,2,3)`
Fundamental Levi-Civita tensor $\epsilon_{i_1i_2i_3}$	`Epsilon(1,2,3)`
Antifundamental Levi-Civita tensor $\epsilon^{\bar{\imath}_1\bar{\imath}_2\bar{\imath}_3}$	`EpsilonBar(1,2,3)`
Sextet representation matrices $(T_6^{a_1})^{\bar{\beta}_3}{}_{\alpha_2}$	`T6(1,2,3)`
Sextet Clebsch-Gordan coefficient $(K_6)^{\bar{\imath}_2\bar{\jmath}_3}{}_{\alpha_1}$	`K6(1,2,3)`
Antisextet Clebsch-Gordan coefficient $(\bar{K}_6)^{\bar{\alpha}_1}{}_{i_2j_3}$	`K6Bar(1,2,3)`

(2.1), and are represented inside an UFO module as,

$$(C_0^{a_1a_2a_3a_4}, C_1^{a_1a_2a_3a_4}, C_2^{a_1a_2a_3a_4}) \leftrightarrow \texttt{['f(1,2,-1) * f(-1,3,4)', ...]}\,,$$
$$(L_0^{\mu_1\mu_2\mu_3\mu_4}, L_1^{\mu_1\mu_2\mu_3\mu_4}, L_2^{\mu_1\mu_2\mu_3\mu_4}) \leftrightarrow \texttt{[L.VVVV1, L.VVVV2, L.VVVV3]}\,.$$

Each color tensor is given as a string representing a polynomial combination of elementary color tensors, whose arguments are integer numbers referring to the position of the particle in the list `particle`. If two indices are contracted, they are represented by a negative integer. The set of all the elementary color tensors currently included in the UFO format, together with the corresponding PYTHON syntax, is given in Table 5. Using these conventions, the color tensors related to the four gluon vertex are given by

$$f^{a_1a_2b}\,f^{ba_3a_4} \quad \leftrightarrow \quad \texttt{'f(1,2,-1) * f(-1,3,4)'}\,,$$
$$f^{a_3a_2b}\,f^{ba_2a_4} \quad \leftrightarrow \quad \texttt{'f(1,3,-1) * f(-1,2,4)'}\,,$$
$$f^{a_1a_4b}\,f^{ba_2a_3} \quad \leftrightarrow \quad \texttt{'f(1,4,-1) * f(-1,2,3)'}\,.$$

Since the list of color tensors associated with a vertex is a mandatory argument of the `Vertex` object, we define the trivial color structure associated with an interaction among non-colored particles as the color tensor `'1'`.

Spin structures such as those appearing in the vertex decompositions in color $\otimes$ spin space are implemented as instances of the class `Lorentz`. All the structures necessary for the whole model are declared in the `lorentz.py` file and we must hence issue at the beginning of the file `vertex.py` the instruction

Table 6: Elementary Lorentz structures

Charge conjugation matrix: $C_{i_1 i_2}$	`C(1,2)`
Epsilon matrix: $\epsilon^{\mu_1\mu_2\mu_3\mu_4}$	`Epsilon(1,2,3,4)`
Dirac matrices: $(\gamma^{\mu_1})_{i_2 i_3}$	`Gamma(1, 2, 3)`
Fifth Dirac matrix: $(\gamma^5)_{i_1 i_2}$	`Gamma5(1,2)`
(Spinorial) Kronecker delta: $\delta_{i_1 i_2}$	`Identity(1,2)`
Minkowski metric: $\eta_{\mu_1\mu_2}$	`Metric(1,2)`
Momentum of the N^{th} particle: $p_N^{\mu_1}$	`P(1,N)`
Right-handed chiral projector: $\left(\frac{1+\gamma 5}{2}\right)_{i_1 i_2}$	`ProjP(1,2)`
Left-handed chiral projector $\left(\frac{1-\gamma 5}{2}\right)_{i_1 i_2}$	`ProjM(1,2)`
Sigma matrices: $(\sigma^{\mu_1\mu_2})_{i_3 i_4}$	`Sigma(1,2,3,4)`

```
import lorentz as L
```

Hence, the `Lorentz` objects used in `vertex.py`, declared in the `lorentz.py` PYTHON module, are preceded by the prefix `L`. As illustrated in the example of the four-gluon vertex, the `lorentz` attribute of the `Vertex` class contains the list of the relevant structures. A `Lorentz` object is instantiated as

```
FFV1 = Lorentz(name = 'FFV1',
               spins = [ 2, 2, 3 ],
               structure = 'Gamma(3,2,1)')
```

All attributes are mandatory. While the attribute `name` is defined in the usual way, the attribute `spins` contains the list of the values of the spins, written as $(2s + 1)$, of the particles entering the vertex. The last argument, `structure`, gives the analytical formula of the Lorentz structure as a string. The conventions for the spin indices is similar to the convention for the color indices: a positive integer i points to the entry i in the list `spins` while negative integers are contracted indices. By default, all the Lorentz indices are supposed to be upper indices, and repeated Lorentz indices are contracted using the Minkowski metric. The list of all objects that can be used to define a Lorentz structure is given in Table 6.

For a given vertex, the G_{ij} quantities appearing in Eq. (2.1) are the 'coordinates' corresponding to the decomposition of a vertex into the color $\otimes$ spin

basis. The `couplings` attribute of the `Vertex` class contains hence a PYTHON dictionary relating the coordinate (i, j) to a `Coupling` object, declared in the file `couplings.py`,

$$G_{ij} \quad \leftrightarrow \quad \texttt{(i,j):C.GC\_1} \ ,$$

By convention, only non-vanishing coordinates G_{ij} are included in this dictionary. Moreover, the `Coupling` objects must be imported at the beginning of the `vertices.py` file through the command

```
import couplings as C
```

The declaration of the `Coupling` objects in the file `couplings.py` is similar to the one of internal parameters. Going back to the example of the four-gluon vertex in Eq. (2.3), the coupling `GC_1` is defined by

```
GC_1 = Coupling(name = 'GC_1',
                value = 'complex(0,1)*G**2',
                order = {'QCD':2}
                )
```

The attribute `value` is a string giving the algebraic expression of the coupling in terms of internal and/or external parameters. The last attribute of a `Coupling` object, `order`, is a PYTHON dictionary where the key for each entry is a string and the value a non-negative integer. In the example above, this means that the four-gluon vertex is proportional to two powers of the strong coupling. This feature allows certain matrix-element generators to generate only subclasses of Feynman diagrams at runtime. This subclass is determined by giving an upper limit for a given interaction type, specified by the key in the dictionary `order`. This concept, together with its implementation into the UFO format, is explained in the next section.

2.5 Controlling various types of couplings in a perturbative expansion

In this section we discuss how to control the different types of expansion parameters that might appear in a perturbative expansion. To illustrate this concept, let us consider the production of a weak boson in association with jets at a hadron collider, *e.g.*, $p\,p \to Z + 4$ jets. This process is dominated by QCD production, while diagrams involving off-shell weak boson exchanges are highly suppressed. In order to speed up the event generation for this process, it is thus desirable to focus exclusively on the strong production of the additional four jets, neglecting all Feynman diagrams with weak boson exchanges. In other words, we would like to select the subset of all the diagrams contributing to the process $p\,p \to Z + 4$ jets that involve at most one electroweak vertex, *i.e.*, at most one power of the electromagnetic coupling constant e.

This can be achieved using tags that allow to count the number of couplings of a given type present in a diagram. In the previous section, we have introduced the `order` attribute of the `Coupling` class. As examples, the `order` of g_s^2 was hence defined as `{'QCD', 2}`, whilst the one of e^2 reads `{'QED':2}`. In the case of the generation of the Feynman diagrams associated to the $p\,p \to Z + 4$ jets process, the coupling order feature allows to restrict the number of couplings of type `QED` to be at most one, neglecting in this way the electroweak production of any additional jet[3]. For certain models, it can be useful to specify a default behavior for some types of coupling orders. This can be done using the `CouplingOrder` class, which we describe in the rest of this section.

Coupling orders are instances of the class `CouplingOrder` and are instantiated in the file `coupling_orders.py`. As a first simple examples, let us consider the coupling orders `QCD` and `QED`, corresponding to the coupling constants g_s and e, respectively. The definitions in `coupling_orders.py` read

```
QCD = CouplingOrder(name = 'QCD',
                    expansion_order = 99,
                    hierarchy = 1
                    )

QED = CouplingOrder(name = 'QED',
                    expansion_order = 99,
                    hierarchy = 2
                    )
```

The class `CouplingOrder` has two mandatory attributes, apart from the ubiquitous `name` attribute. First, the attribute `expansion_order` is an integer specifying the maximal number of couplings of this type that should be included in a given process. The default value is 99, indicating that any number is allowed. The second attribute, `hierarchy`, is an integer that allows one to classify different types of interactions according to their relative strength. In the above example, we have `QCD.hierarchy` $= 1$ and `QED.hierarchy` $= 2$, reflecting the fact that g_s^4 is of the same order of magnitude as e^2. The `CouplingOrder` objects then allow certain matrix-element generators to define a default behavior for the maximal number of couplings of a given type that can appear in a diagram, based on the upper bound set by `expansion_order` and the relative strength among the various couplings.

[3] We stress that coupling orders are a property of the matrix-element generators, *i.e.*, the matrix-element generator in question needs to support this feature to use it.

3 The FeynRules UFO interface

Even though it is possible to implement a model into the UFO format by hand, this procedure can be a tedious and error-prone task, because all the vertices need to be entered one at the time. In order to alleviate this problem, we have implemented an interface into FeynRules that allows one to export a given model directly in the UFO format. The FeynRules model contains, on the one hand, basic model information (such as the particle content or the parameters of the model) which is implemented as described in Refs. [19,20,22]. In particular, a new feature of the FeynRules model files allows to specify the hierarchy between the different types of couplings and the limit up to which they should appear in the perturbative expansion (see Section 2.5). This is achieved by including the global variables `M$InteractionOrderHierarchy` and `M$InteractionOrderLimit` directly into the FeynRules model file[4]. Considering the example of the types of couplings QED and QCD presented in Section 2.5, the FeynRules model implementation includes then the definition

```
M$InteractionOrderHierarchy = {
  {QCD, 1},
  {QED, 2}
}

M$InteractionOrderLimit = {
  {QCD, 99},
  {QED, 99}
}
```

Note that this new feature is optional for each type of coupling. If a given type if not represented in one of the two lists, the default values assigned will be 1 for the `hierarchy` and 99 for the `expansion_order`.

The FeynRules UFO interface can be called in exactly the same way as all the other FeynRules interfaces,

`WriteUFO[` $\mathcal{L}_1, \mathcal{L}_2, \ldots,$ *options* `]`

where $\mathcal{L}_1, \mathcal{L}_2, \ldots$ denote the Lagrangians of the model, and *options* denotes a set of options supported by the interface. The interface shares all the options of the function `FeynmanRules[ ]`, plus some additional options summarized in Table 7. When this command is issued, FeynRules internally calls the function `FeynmanRules[ ]` to compute all the vertices associated with the

[4] `InteractionOrder` is the FeynRules equivalent to the `order` attribute of the UFO `Coupling` object presented in Section 2.4.

Table 7: Additional options of the function `WriteUFO`

`Input`	A list of vertices computed previously and to be included into the UFO output.
`Output`	A string, the name of the output directory. The default is the value of the FeynRules variables `M$ModelName` with `_UFO` appended.
`DialogBox`	If `Off`, no dialog boxes open up when running the interface. The default is `On`.

Lagrangians $\mathcal{L}_i$. After the complete list of Feynman rules has been obtained, the vertices are decomposed into a color $\otimes$ spin basis [5] according to Eq. (2.1), and the different `Lorentz` and `Coupling` objects are identified. At the end of the procedure, all the information about the model is written to files according to the format presented in Section 2, and saved in a directory called `*_UFO`, where `*` denotes the name of the model.

Note that there is a crucial difference between the UFO interface and the other existing interfaces included in the FeynRules package. While all other interfaces select the subset of vertices that are supported by the matrix-element generators (in general, this subset consists more or less into renormalizable operators) while rejecting all other vertices, the UFO interface is completely agnostic of the matrix-element generator, and hence does not make any assumptions on whether a given generator accepts a certain vertex structure. The UFO output will hence always contain *all* the vertices of the model, and it is then up to the matrix-element generator to assure that only allowed vertices are processed.

4 The UFO format beyond tree level

During the last five years, a lot of progress has been made in the automation of the computation of next-to-leading order matrix elements, both regarding the generation of the real corrections with the appropriate subtraction terms [31,32,33,34,35,36], and the development of algorithms for calculating loop amplitudes numerically [37,38,39,40,41,42].

Although currently the focus of the UFO is to provide a common input for tree-level Monte Carlo programs, the format is by no means restricted

[5] Note that this decomposition might not be unique.

to tree-level generators only. Hence, the one-loop matrix-element generator GoSam [24,25,26] contains an interface to the UFO format, where the information from the Python module described in Section 2 is translated into a model definition for the QGraf package [43] together with a Form [44] module substituting the expressions from the Feynman rules. This setup has been successfully applied to simple one-loop calculations in the Minimal Supersymmetric Standard Model, where the renormalization can still be worked out by hand. For more involved computations, however, one would like to automate not only the calculation of the matrix elements but also the derivation of the counterterms associated with a given renormalization procedure. Although FeynRules in its current version does not yet support the calculation of renormalization constants and counterterms, we propose in this section a generic prescription for their inclusion in the UFO format.

Assuming a multiplicative renormalization prescription, the relation between bare and renormalized quantities is given by $m_0 = Z_m m_r = (1 + \delta Z_m) m_r$, where m represents a generic parameter, and by $\Psi_0 = Z_\Psi^{1/2} \Psi_r = (1 + \frac{1}{2}\delta Z_\Psi)\Psi_r$ for the fields. The general case of propagator mixing allows the last equation to take a matrix form. Furthermore, it is assumed that the ultraviolet divergences have been regularized dimensionally, the renormalization constants being thus expressed as Laurent series in $\epsilon = (4-D)/2$ where D is the number of space-time dimensions. Taking into account that the format should not be restricted to one type of perturbative corrections but should be extendable to any $\alpha_s^{n_1}\alpha_{EW}^{n_2}$ order of the perturbative expansion, we can make the ansatz

$$\delta Z_i = \sum_{n_1,n_2=1}^{\infty} \frac{\alpha_s^{n_1}\alpha_{EW}^{n_2}}{(2\pi)^{n_1+n_2}} \sum_{p=-\infty}^{\infty} z_{n_1,n_2}^{(p)}\, \epsilon^p, \tag{4.1}$$

where, in general, only a small subset of the coefficients $z_{n_1,n_2}^{(p)}$ is non-zero. To include the renormalization constants associated with a parameter, we propose to add an attribute to the `Parameter` class denoted `counterterm`. Taking the example of the strong coupling constant introduced in Section 2.3, `G`, its definition is augmented by

```
G.counterterm = { (1,0): {-1: '2./3*NF*TF-11./6*CA'} }
```

in order to include the QCD one-loop effects on the strong coupling constant. The renormalization constant is represented by a Python dictionary where the keys are the pairs (n_1, n_2) introduced in Eq. (4.1) and the values are the Laurent series in ϵ. The latter are represented by dictionaries with the powers of ϵ as keys and strings representing Python expressions as values. Let us note that the symbols `NF`, `TF` and `CA` which have been introduced must be either replaced by their proper values or be defined as model parameters. For models containing more than two coupling constants, the pairs (n_1, n_2) are replaced by the corresponding n-tuples. Similarly, wave function renormaliza-

tion constants [6] are included in the `counterterm` attribute which is added to the `Particle` class. It contains the object δZ_Ψ, implemented following a structure identical to the one described for the `Parameter` class.

In addition to the renormalization constants, one also needs analytical expressions for the counterterms. In general, they can be described as vertices, starting from two-point vertices for the propagator counterterms, which are included in the files `ctvertices.py` and `ctcouplings.py`. Analogously, the initialization file `__init__.py` contains, in addition to the lists described in the previous section, the lists `all_ctvertices` and `all_ctcouplings`. Similarly to Section 2.4, the file `ctvertices.py` contains all the counterterm vertices represented by objects of the `Vertex` class, and the related couplings are included in the file `ctcouplings.py`. These couplings reflect the nature of the renormalization constants as Laurent expansion in ϵ. Using the generic structure for vertices presented in Eq. (2.1) [7], we can write a counterterm coupling as

$$G_{ij} = g_{ij}^{(0)} \sum_{n_1,n_2=1}^{\infty} \frac{\alpha_s^{n_1} \alpha_{EW}^{n_2}}{(2\pi)^{n_1+n_2}} \sum_{p=-\infty}^{\infty} c_{ij,n_1,n_2}^{(p)} \, \epsilon^p \ . \tag{4.2}$$

This ansatz allows for some freedom with respect to numeric factors that can be part of either $g_{ij}^{(0)}$ or c_{ij,n_1,n_2}. However, the power of the coupling constants in $g_{ij}^{(0)}$ must correspond to the one included in the associated tree level vertex. Hence, the counterterm coupling can be easily declared using the `Coupling` class,

```
GCT_1 = Coupling(name = 'GCT_1',
       value = 'complex(0,1)*G**2',
       counterterm = {(1,0): {-1: G.counterterm}},
       order = {'QCD':2}
   )
```

The prefactor $g_{ij}^{(0)}$ is stored in the `value` attribute whereas all relative corrections $c_{ij,n_1,n_2}^{(p)}$ are mapped to the attribute `counterterm`, using the same philosophy as in the case of the classes `Parameter` and `Particle`. Finally, the attribute `order` reflects the interaction order of $g_{ij}^{(0)}$ and does not take into account the additional powers of coupling constants coming from the sum over n_1 and n_2.

The amendments described in this section transmit all information necessary for an efficient computation of ultraviolet counterterms by a matrix-element generator. Furthermore, the same approach could be used in order to include

[6] Mixing of on-shell particles is assumed to be zero. However, in propagators, mixing is realized through two-point vertices.

[7] The basis in the color $\otimes$ spin space associated to a counterterm vertex might be different from the corresponding tree-level one.

other counterterm-like objects, such as the rational R_2 terms [45,46,47,48] in the OPP approach [49,50].

5 An example

An UFO model contains the full set of vertices of a model, *i.e.*, all the Lorentz and color structures appearing in all the vertices together with their coefficients. Consequently, it is also suited for models with Lorentz structures that are not SM-like, a characteristic shared by all models with higher-dimensional operators. In the following, we illustrate the UFO format on the example of the Strongly Interacting Light Higgs (SILH) model [51]. The SILH model is an effective theory describing the interactions of the Higgs boson considering it as the Goldstone boson linked to a new strongly interacting sector. Since it is already implemented in FeynRules [20], this model can easily be exported to the Monte Carlo tools via the corresponding FeynRules interfaces.

The particle content of the SILH model is the same as in the SM. The particularities of the model come solely from the new interactions induced by dimension-six operators involving SM fields. In this short example, we focus on the decay of the Higgs boson H into two W-bosons. The non-SM part of the SILH Lagrangian affecting this decay rate reads

$$\begin{aligned}\mathcal{L}_{\text{SILH}}^{HWW} = &\frac{c_H}{2f^2}\partial^\mu\left(H^\dagger H\right)\partial_\mu\left(H^\dagger H\right) + \frac{ic_W g}{2g_\rho^2 f}\left(H^\dagger\sigma^i\overleftrightarrow{D^\mu}H\right)(D^\nu W_{\mu\nu})^i \\ &+\frac{ic_{HW}g}{16\pi^2 f^2}(D^\mu H)^\dagger\sigma^i(D^\nu H)W^i_{\mu\nu}\end{aligned} \tag{5.1}$$

where f is the suppression scale for the new operators, g and g_ρ are the coupling constants of the weak and the new strong interaction, respectively, and c_H, c_W and c_{HW} are free coefficients. In the expression above, we have introduced the covariant derivative D_μ (taken in the appropriate representation), the W-boson field strength tensor $W_{\mu\nu}$, and the Pauli matrices σ^i. The effective Lagrangian has been obtained after an expansion in $1/f$ up to $\mathcal{O}(1/f^2)$. Hence, the HWW vertex reads now

$$\begin{aligned}ig_w^2\left[\frac{v}{2}\left(1-c_H\frac{\xi}{2}\right)\eta_{\mu_2,\mu_3} + c_{HW}\xi\frac{p_1^{\mu_2}p_2^{\mu_3}+p_1^{\mu_3}p_3^{\mu_2}-(p_1.p_2+p_1.p_3)\,\eta_{\mu_2,\mu_3}}{32\pi^2 v}\right. \\ \left.+c_W\xi\frac{(p_2.p_2+p_3.p_3)\,\eta_{\mu_2,\mu_3}-p_2^{\mu_2}p_2^{\mu_3}-p_3^{\mu_2}p_3^{\mu_3}}{2vg_\rho^2}\right]\end{aligned} \tag{5.2}$$

where $\xi = \frac{v^2}{f^2}$, v being the vacuum expectation value of the neutral component of the Higgs doublet, and p_i are the momenta of the interacting particles.

After an UFO implementation of the model via the corresponding FeynRules interface has been obtained, this vertex appears in `vertices.py` as

```
V_22 = Vertex(name = 'V_22',
              particles = [ P.W__minus__, P.W__plus__, P.H ],
              color = [ '1' ],
              lorentz = [ L.VVS1, L.VVS5, L.VVS8 ],
              couplings = {(0,1):C.GC_56,(0,2):C.GC_59,
                           (0,0):[ C.GC_30, C.GC_68 ]}).
```

The color tensor is trivial since all particles are color singlets. On the contrary, the spin structure is more complicated because of the non-trivial tree-level Lorentz structures of Eq.(5.2). As an example, the `VVS8` spin tensor is defined in the file `lorentz.py` as

```
VVS8 = Lorentz(name = 'VVS8',
               spins = [ 3, 3, 1 ],
               structure = 'P(1,3)*P(2,1) + P(1,2)*P(2,3)
                           - P(-1,1)*P(-1,3)*Metric(1,2)
                           - P(-1,2)*P(-1,3)*Metric(1,2)')
```

The product of `VVS8` and `GC_59` corresponds to the second term of Eq. (5.2). The coupling order of the coupling is given by `NP=1`, indicating that it contains one power of ξ. It is important to note that `VVS1`, the Lorentz structure of the first term in Eq.(5.2) is associated with two coefficients, split according to their coupling order. In particular, `GC_30` is the SM part and corresponds to the order `QED=1`, while `GC_68` is the new physics contribution proportional to c_H and of order `NP=1`. Interferences between SM and new physics operators can be extracted from the interaction order of the vertex. Indeed, due to our choice for the c_i coefficients, the new physics pieces of the vertex have an interaction order equal to `NP=1` and `QED=0`. Therefore, the interference is obtained by computing the difference between all the contributions (`NP=1 QED=1`) and the pure SM (`NP=0 QED=1`) and SILH (`NP=1 QED=0`) ones.

The Lagrangian of Eq. (5.1) is truncated at order $\mathcal{O}(1/f^2)$, or equivalently at order $\mathcal{O}(\xi)$. Computation of matrix elements at higher order in ξ would hence not be reliable without adding the corresponding terms in the expansion of the Lagrangian. For instance, the production of a Higgs-boson pair by weak boson fusion involves a diagram containing two vertices with order `NP=1`, as presented in Fig. 1, but also an additional diagram related to the expansion of the Lagrangian at order $\mathcal{O}(\xi^2)$, which is absent from our SILH model implementation. To prevent the user from such issues, the model builder should warn him that $\mathcal{O}(\xi^n)$ amplitudes, with $n \geq 2$, cannot be in general computed using the implemented Lagrangian, and that the `NP` order should be at most equal to one. This restriction is easily included in the UFO model through to

Fig. 1. Feynman diagram contributing to $W^+W^- \to H\,H$ at $\mathcal{O}(\xi^2)$. The shaded boxes represents the $\mathcal{O}(\xi)$ part of the HWW vertex in Eq. (5.2).

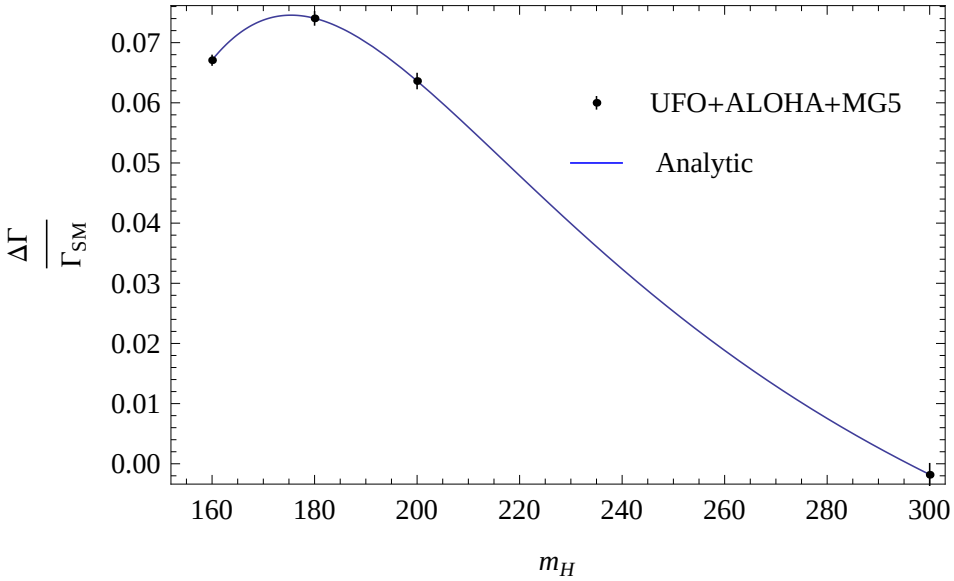

Fig. 2. The relative correction $\frac{\Delta\Gamma}{\Gamma_{SM}} \equiv \frac{\Gamma_{SILH}-\Gamma_{SM}}{\Gamma_{SM}}$ to the decay width of the Higgs boson into two W bosons in the SILH model for $g_\rho = 1$, $f = 1$ TeV, $\xi = 0.060623$, $c_H = 4$, $c_W = 2$ and $c_{HW} = 800$. Only the interference terms between SM and diagrams involving the new operators are taken into account in Γ_{SILH} (see Eq. (6.6) of Ref. [20] and references therein).

the `expansion_order` argument of the `CouplingOrder` object

```
NP = CouplingOrder(name = 'NP',
                   expansion_order = 1,
                   hierarchy = 2
                   )
```

The `hierarchy` argument being set to 2 ensures that the new physics contributions are not removed by default in the weak processes.

We choose to validate our UFO model by using the computation tools MadGraph version 5 [13], which thanks to the Aloha module [52], allows for a full support of the higher dimensional operators. We have computed the Higgs partial decay width into a W-boson pair, using a very large value for c_{HW} in order to render the associated new physics contribution dominant and to subsequently test the treatment of the higher-dimensional operators by MadGraph. Moreover, this contribution is not proportional to the SM result, contrary to the others. In Fig. 2, we confront the results to hand-made analytical calculations and found perfect agreement.

6 Conclusion

In this paper, we have presented a new model format for matrix-element generators, the Universal FeynRules Output (UFO) format. While most of the present generators have implicit assumptions on the color and/or Lorentz structures appearing in the different interaction vertices of a given model, the UFO format has been designed to go beyond these constraints, by being agnostic of any, even unforeseen, restrictions. Indeed, unlike the more traditional table-based model formats (as used by many Monte Carlo codes), the UFO represents all the information about a model terms of abstract Python classes that can accommodate any (reasonable) particle physics model. As an example, despite the fact that so far only color singlet, triplet, sextet and octet particles have been implemented into the UFO format, the extension to more exotic representations of the QCD gauge group is in principle straightforward, without requiring any change to the UFO format itself. A similar change would be very hard to perform in some of the existing table-based model formats. Finally, we emphasize that the format gives a full support to Les Houches accord conventions for model parameters and we also illustrate how it could be extented in the context of the next-to-leading order tools, including, *e.g.*, counterterms and the so-called R_2 terms. Presently, the UFO format is already used by the MadGraph version 5 and GoSam generators and will be used in a near future by Herwig++.

Acknowledgments

The authors are grateful to Priscila de Aquino, Neil Christensen, Will Link and to the whole MG5 development team for useful and constructive discussions. ClD and BF are grateful to the CP3 Louvain for the hospitality at various stages during this project. CeD is a Research Fellow of the 'Fonds National de la Recherche Scientifique' (FNRS), Belgium. OM is 'Chercheur scientifique logistique postdoctoral F.R.S-FNRS', Belgium. TR is supported by the Humboldt Foundation, in the framework of the Sofja Kovaleskaja Award Project "Advanced Mathematical Methods for Particle Physics", endowed by the German Federal Ministery of Education and Research. This work was partially supported by the Theory-LHC France Initiative, by the Research Executive Agency (REA) of the European Union under the Grant Agreement number PITN-GA-2010-264564 (LHCPhenoNet), by the Belgian Federal Office for Scientific, Technical and Cultural Affairs through the Interuniversity Attraction Poles Program - Belgium Science Policy P6/11-P and by the ISN MadGraph convention 4.4511.10.

References

[1] M. L. Mangano, M. Moretti, F. Piccinini, R. Pittau, A. D. Polosa, ALPGEN, a generator for hard multiparton processes in hadronic collisions, JHEP 07 (2003) 001. arXiv:hep-ph/0206293.

[2] T. Gleisberg, S. Hoche, Comix, a new matrix element generator, JHEP 12 (2008) 039. arXiv:0808.3674, doi:10.1088/1126-6708/2008/12/039.

[3] A. Pukhov, et al., CompHEP: A package for evaluation of Feynman diagrams and integration over multi-particle phase space. User's manual for version 33, arXiv:hep-ph/9908288.

[4] E. Boos, et al., CompHEP 4.4: Automatic computations from Lagrangians to events, Nucl. Instrum. Meth. A534 (2004) 250–259. arXiv:hep-ph/0403113, doi:10.1016/j.nima.2004.07.096.

[5] A. Pukhov, CalcHEP 3.2: MSSM, structure functions, event generation, batchs, and generation of matrix elements for other packages, arXiv:hep-ph/0412191.

[6] A. Cafarella, C. G. Papadopoulos, M. Worek, Helac-Phegas: a generator for all parton level processes, Comput. Phys. Commun. 180 (2009) 1941–1955. arXiv:0710.2427, doi:10.1016/j.cpc.2009.04.023.

[7] G. Corcella, et al., HERWIG 6: An event generator for hadron emission reactions with interfering gluons (including supersymmetric processes), JHEP 01 (2001) 010. arXiv:hep-ph/0011363.

[8] M. Bahr, S. Gieseke, M. Gigg, D. Grellscheid, K. Hamilton, et al., Herwig++ Physics and Manual, Eur.Phys.J. C58 (2008) 639–707. arXiv:0803.0883, doi:10.1140/epjc/s10052-008-0798-9.

[9] T. Stelzer, W. F. Long, Automatic generation of tree level helicity amplitudes, Comput. Phys. Commun. 81 (1994) 357–371. arXiv:hep-ph/9401258, doi:10.1016/0010-4655(94)90084-1.

[10] F. Maltoni, T. Stelzer, MadEvent: Automatic event generation with MadGraph, JHEP 02 (2003) 027. arXiv:hep-ph/0208156.

[11] J. Alwall, et al., MadGraph/MadEvent v4: The New Web Generation, JHEP 09 (2007) 028. arXiv:0706.2334.

[12] J. Alwall, et al., New Developments in MadGraph/MadEvent, AIP Conf. Proc. 1078 (2009) 84–89. arXiv:0809.2410, doi:10.1063/1.3052056.

[13] J. Alwall, M. Herquet, F. Maltoni, O. Mattelaer, T. Stelzer, MadGraph 5 : Going Beyond, JHEP 1106 (2011) 128. arXiv:1106.0522, doi:10.1007/JHEP06(2011)128.

[14] T. Gleisberg, et al., SHERPA 1.alpha, a proof-of-concept version, JHEP 02 (2004) 056. arXiv:hep-ph/0311263.

[15] T. Gleisberg, et al., Event generation with SHERPA 1.1, JHEP 02 (2009) 007. arXiv:0811.4622, doi:10.1088/1126-6708/2009/02/007.

[16] M. Moretti, T. Ohl, J. Reuter, O'Mega: An Optimizing matrix element generator, arXiv:hep-ph/0102195.

[17] W. Kilian, T. Ohl, J. Reuter, WHIZARD: Simulating Multi-Particle Processes at LHC and ILC, Eur. Phys. J. C71 (2011) 1742. arXiv:0708.4233, doi:10.1140/epjc/s10052-011-1742-y.

[18] A. Semenov, LanHEP: A package for automatic generation of Feynman rules from the Lagrangian, Comput. Phys. Commun. 115 (1998) 124–139. doi:10.1016/S0010-4655(98)00143-X.

[19] N. D. Christensen, C. Duhr, FeynRules - Feynman rules made easy, Comput. Phys. Commun. 180 (2009) 1614–1641. arXiv:0806.4194, doi:10.1016/j.cpc.2009.02.018.

[20] N. D. Christensen, P. de Aquino, C. Degrande, C. Duhr, B. Fuks, et al., A Comprehensive approach to new physics simulations, Eur.Phys.J. C71 (2011) 1541. arXiv:0906.2474, doi:10.1140/epjc/s10052-011-1541-5.

[21] N. D. Christensen, C. Duhr, B. Fuks, J. Reuter, C. Speckner, Exploring the golden channel for HEIDI models using an interface between WHIZARD and FeynRules, arXiv:1010.3251.

[22] C. Duhr, B. Fuks, A superspace module for the FeynRules package, Comput. Phys. Commun. 182 (2011) 2404–2426. arXiv:1102.4191, doi:10.1016/j.cpc.2011.06.009.

[23] T. Hahn, Generating Feynman diagrams and amplitudes with FeynArts 3, Comput. Phys. Commun. 140 (2001) 418–431. arXiv:hep-ph/0012260, doi:10.1016/S0010-4655(01)00290-9.

[24] P. Mastrolia, G. Ossola, T. Reiter, F. Tramontano, Scattering AMplitudes from Unitarity-based Reduction Algorithm at the Integrand-level, JHEP 1008 (2010) 080. arXiv:1006.0710, doi:10.1007/JHEP08(2010)080.

[25] G. Heinrich, G. Ossola, T. Reiter, F. Tramontano, Tensorial Reconstruction at the Integrand Level, JHEP 1010 (2010) 105. arXiv:1008.2441, doi:10.1007/JHEP10(2010)105.

[26] G. Cullen, N. Greiner, G. Heinrich, G. Luisoni, P. Mastrolia, G. Ossola, T. Reiter, F. Tramontano, Automated One-Loop Calculations with GoSam, arXiv:1111.2034.

[27] K. Nakamura, et al., Review of particle physics, J.Phys.G G37 (2010) 075021. doi:10.1088/0954-3899/37/7A/075021.

[28] P. Z. Skands, B. Allanach, H. Baer, C. Balazs, G. Belanger, et al., SUSY Les Houches accord: Interfacing SUSY spectrum calculators, decay packages, and event generators, JHEP 0407 (2004) 036. arXiv:hep-ph/0311123, doi:10.1088/1126-6708/2004/07/036.

[29] B. Allanach, C. Balazs, G. Belanger, M. Bernhardt, F. Boudjema, et al., SUSY Les Houches Accord 2, Comput.Phys.Commun. 180 (2009) 8–25. arXiv:0801.0045, doi:10.1016/j.cpc.2008.08.004.

[30] J. Alwall, E. Boos, L. Dudko, M. Gigg, M. Herquet, et al., A Les Houches Interface for BSM Generators, arXiv:0712.3311.

[31] T. Gleisberg, F. Krauss, Automating dipole subtraction for QCD NLO calculations, Eur. Phys. J. C53 (2008) 501–523. arXiv:0709.2881, doi:10.1140/epjc/s10052-007-0495-0.

[32] M. H. Seymour, C. Tevlin, TeVJet: A general framework for the calculation of jet observables in NLO QCD, arXiv:0803.2231.

[33] K. Hasegawa, S. Moch, P. Uwer, Automating dipole subtraction, Nucl. Phys. Proc. Suppl. 183 (2008) 268–273. arXiv:0807.3701, doi:10.1016/j.nuclphysbps.2008.09.115.

[34] R. Frederix, T. Gehrmann, N. Greiner, Automation of the Dipole Subtraction Method in MadGraph/MadEvent, JHEP 09 (2008) 122. arXiv:0808.2128, doi:10.1088/1126-6708/2008/09/122.

[35] M. Czakon, C. G. Papadopoulos, M. Worek, Polarizing the Dipoles, JHEP 08 (2009) 085. arXiv:0905.0883, doi:10.1088/1126-6708/2009/08/085.

[36] R. Frederix, S. Frixione, F. Maltoni, T. Stelzer, Automation of next-to-leading order computations in QCD: the FKS subtraction, JHEP 10 (2009) 003. arXiv:0908.4272, doi:10.1088/1126-6708/2009/10/003.

[37] G. Zanderighi, Recent theoretical progress in perturbative QCD, arXiv:0810.3524.

[38] R. K. Ellis, K. Melnikov, G. Zanderighi, Generalized unitarity at work: first NLO QCD results for hadronic W^+ 3jet production, JHEP 04 (2009) 077. arXiv:0901.4101, doi:10.1088/1126-6708/2009/04/077.

[39] C. F. Berger, et al., Precise Predictions for W + 3 Jet Production at Hadron Colliders, Phys. Rev. Lett. 102 (2009) 222001. arXiv:0902.2760, doi:10.1103/PhysRevLett.102.222001.

[40] A. van Hameren, C. G. Papadopoulos, R. Pittau, Automated one-loop calculations: a proof of concept, JHEP 09 (2009) 106. arXiv:0903.4665, doi:10.1088/1126-6708/2009/09/106.

[41] C. F. Berger, et al., Precise Predictions for W + 4 Jet Production at the Large Hadron Collider, Phys. Rev. Lett. 106 (2011) 092001. arXiv:1009.2338, doi:10.1103/PhysRevLett.106.092001.

[42] V. Hirschi, R. Frederix, S. Frixione, M. V. Garzelli, F. Maltoni, et al., Automation of one-loop QCD corrections, JHEP 1105 (2011) 044. arXiv:1103.0621, doi:10.1007/JHEP05(2011)044.

[43] P. Nogueira, Automatic Feynman graph generation, J.Comput.Phys. 105 (1993) 279–289. doi:10.1006/jcph.1993.1074.

[44] J. Vermaseren, New features of FORM, arXiv:math-ph/0010025.

[45] G. Ossola, C. G. Papadopoulos, R. Pittau, On the Rational Terms of the one-loop amplitudes, JHEP 0805 (2008) 004. arXiv:0802.1876, doi:10.1088/1126-6708/2008/05/004.

[46] P. Draggiotis, M. Garzelli, C. Papadopoulos, R. Pittau, Feynman Rules for the Rational Part of the QCD 1-loop amplitudes, JHEP 0904 (2009) 072. arXiv:0903.0356, doi:10.1088/1126-6708/2009/04/072.

[47] M. Garzelli, I. Malamos, R. Pittau, Feynman rules for the rational part of the Electroweak 1-loop amplitudes, JHEP 1001 (2010) 040. arXiv:0910.3130, doi:10.1007/JHEP01(2010)040,10.1007/JHEP10(2010)097.

[48] M. Garzelli, I. Malamos, R. Pittau, Feynman rules for the rational part of the Electroweak 1-loop amplitudes in the R_xi gauge and in the Unitary gauge, JHEP 1101 (2011) 029. arXiv:1009.4302, doi:10.1007/JHEP01(2011)029.

[49] G. Ossola, C. G. Papadopoulos, R. Pittau, Reducing full one-loop amplitudes to scalar integrals at the integrand level, Nucl.Phys. B763 (2007) 147–169. arXiv:hep-ph/0609007, doi:10.1016/j.nuclphysb.2006.11.012.

[50] G. Ossola, C. G. Papadopoulos, R. Pittau, Numerical evaluation of six-photon amplitudes, JHEP 0707 (2007) 085. arXiv:0704.1271, doi:10.1088/1126-6708/2007/07/085.

[51] G. F. Giudice, C. Grojean, A. Pomarol, R. Rattazzi, The Strongly-Interacting Light Higgs, JHEP 06 (2007) 045. arXiv:hep-ph/0703164, doi:10.1088/1126-6708/2007/06/045.

[52] P. de Aquino, W. Link, F. Maltoni, O. Mattelaer, T. Stelzer, ALOHA: Automatic Libraries Of Helicity Amplitudes for Feynman Diagram Computations, Comput.Phys.Commun. 183 (2012) 2254–2263. arXiv:1108.2041, doi:10.1016/j.cpc.2012.05.004.